The Architectural Metaphor as a Foundation for Systems Engineering

Richard F. Hilliard II
Timothy B. Rice
Stephen C. Schwarm
The MITRE Corporation
Bedford, Massachusetts 01730*

Abstract

Over the history of systems engineering, there have been numerous attempts to establish the foundations of that field on other disciplines. In this paper, we explore a new approach, founding systems engineering on an *architectural metaphor*. This approach is based upon our work of the past five years to establish a discipline of systems architecture. We outline the key principles of our approach and discuss their relevance to the broader field of systems engineering.

Introduction

We now have over forty years of experience in systems engineering, a field which has grown up to address the planning and construction of large, complex, typically information-rich, systems to meet real world needs. In that period, there have been various attempts to discover appropriate foundations for the field, or to ground systems engineering in other disciplines. These have ranged from set theory to systems theory to category theory to psychology [1, 15, 21].

In this paper we explore another approach, that of grounding systems engineering on an *architectural metaphor*, based on previous architectural traditions, especially that of building architecture. For the past five years, we have been evolving a disciplined approach to software systems architecture. In this paper, we will review the principles of our approach and discuss their implications for the wider field of systems engineering.

In our community there is widespread interest in using the concept of architecture to control and manage the procurement and evolution of information systems. The current interest in architecture is motivated by the desire to build our systems "faster, better and cheaper" [9].

Although we are principally interested in *system* architectures, much of current architectural thinking is motivated by the issues of *software* architecture [20]. In our work, we draw freely from both the software and systems engineering communities, making a distinction only when warranted. Since most systems of interest are software-intensive, this is rarely a distinguishing criterion.

A Practical Architecture Method

Since 1990, we have been refining an approach to the specification and analysis of system architectures [11]. This work has several goals:

1. To articulate the foundations of software systems architecture by defining a framework of key terms and concepts;
2. To define practical techniques for modeling and analyzing architectures within that framework;
3. To apply that framework within the context of system acquisition and development to:
   - define standards for architectural description;
   - enable objective evaluation of delivered and proposed architectures; and,
   - provide architects with automated support for the creation and maintenance of system architecture descriptions;

This paper will focus on foundational issues; we have presented our method and experiences elsewhere [6, 7].

Defining "architecture"

Although it is currently quite fashionable to use the term "architecture," there is no widely accepted definition. Intuitively, most people use it to mean roughly the large-scale structure of a system, its major constituents and their interconnections. Beyond that, there is much discussion and variation [8]. We have adopted the following definition which is somewhat different than most:

> An *architecture* is the highest level conception of a system in its environment.

This definition has several consequences:

- Every system has an architecture.
- Systems are situated in their environments. An architecture reflects the *whole* system in response to that environment.
- "Conception" is intentionally abstract – perhaps an idea in someone's head. To put it to engineering use, we need a concrete representation of the architecture. Building architects produce blueprints and other representations – not architectures per se.

Architecture in Context

By recognizing that an architecture pertains to the whole system, our definition excludes a number of current uses of that term. An especially confusing use is prevalent within the defense systems community, where it is fashionable to speak of a system as having multiple architectures: an "operational architecture," a "system architecture" and a "technical architecture" [4]. We might as well describe an office building as having an "occupant architecture," a

* Appears in *Proceedings of the Sixth Annual International Symposium of the International Council on Systems Engineering*, Boston, July 1996

"heating and ventilation architecture," and a "site architecture"! We say an architecture has a number of aspects, such as technical, operational and system concerns. Each may be formalized by defining one or more *views*. It is the architect's job to integrate these aspects into a single architecture.

Within software architecture, there has been a tendency to "lift" techniques from design to apply architecturally; the literature is filled with "functional architectures," "information architectures" [5] and "object architectures" [13]. We believe this tendency reflects a confusion of design entities with architectural goals. We have tried to separate the two in our work.

Recognizing that an architecture reflects a system in its environment, it must address a much wider range of concerns than those of a design. A software design is internally focused on the interfaces, operations, and modules which constitute a software system. The "outside" has been abstracted away during requirements analysis. In approaches which view architecture as a part of high-level design [14, 20], there is a single implied audience: the designers and implementors of the system, relegating others, such as the client, the users, and operators to a subsidiary role.

Whereas a design is internally focused, an architecture is *externally focused* – the architect must be cognizant of the system's environment. Just as a building architect takes into account a project's surroundings, an architecture must be sensitive to the environment of the intended system. For a software system, the environment includes the systems with which the target system will interact or co-exist, the operational concept for that system, and its anticipated users.

Frequently, the term "architecture" is applied to middleware frameworks such as CORBA and DCE. More generally, frameworks, as that term is used in the object-oriented programming community, offer the prospect of large-scale reuse within a domain [12]. A framework is a useful way to implement a class of systems with similar characteristics – but a framework is not an architecture itself, it is rather a building technology applicable to a class of systems. Frameworks are inwardly focused on how a system will be built rather than on the outward intent of the system. For this reason, we prefer to call them *software component frameworks*. An architecture will make specific decisions but not necessarily particular design choices.

Moving outside the internal focus of design, a number of potential audiences for an architecture become relevant, each with potentially different concerns for the system.

Boehm and others have identified a number of stakeholders of a system's architecture [2, 8]. The key stakeholders will vary from system to system, they could include: acquisition agent, client, customer, developer, inspector, maintainer, operator, owner, planner, service provider, user, and vendor. We further distinguish a subset of the stakeholders called *audiences*. Audiences are those stakeholders who would be expected to directly access architectural descriptions. Our approach is therefore "subject-oriented" rather than object-oriented. Its orientation derives from that of the stakeholders for the architecture.

An Architecture Process

We have developed a candidate process for specifying and managing an architecture, shown in table 1. As noted above, this paper will not focus on that process, which has been discussed elsewhere [7]. The purpose of the table is to provide a context for discussing some of the architectural elements which would be applicable to systems engineering.

1. Understand system context
   - Needs Analysis
   - Program cost and schedule
   - Products: needs, goals, vision statements
2. Select views
3. Analyze each view
   - Products: a "blueprint" of each view with associated decisions, rules and open issues
4. Integrate views (for consistency)
5. Trace views to needs (for completeness)
6. Iterate activities 3, 4 and 5
   - Until no more unresolved questions
   - If there are numerous unresolved issues, revisit step 2
7. Validate
   - Against formal requirements
   - Conformance of design and implementation (as built) to architecture (as specified)

Table 1: An Architectural Process

Architectural Needs. As in many branches of engineering, our method begins with understanding the problem and its constraints including the requirements, program cost and schedule demands; i.e., the concerns of the various stakeholders. Our understanding of these is distilled into three products, developed with the client, and ideally "owned" by the client, which capture both the specific requirements for the system, and also the general direction and intent of the client.

For the large command and control and information systems we typically deal with, there are one or more requirements specification documents, often running into hundreds or even thousands of pages. Even when well-written, these formal requirements documents have proven to be too difficult to use effectively to derive and analyze system architectures. We analyze these requirements into a set of architectural *needs*, which capture the general requirements, without unnecessary detail. For a recent information system architecture, we distilled thousands of requirements from several sources into 138 architectural needs. These needs are maintained in a data base to preserve traceability of our architectural decisions back to the full set of formal requirements.

In addition to the "abstracted requirements" reflected by our needs analysis, we prepare two other statements with the client. First, a statement of *goals* reflects the client's evaluation criteria – how the client will evaluate whether the system is a success. Goals include meeting the needs, as well as preferences and desires which do not occur as requirements. Second, we define a *vision* statement. Whereas the goals specify what is to be achieved in the current system, the vision statement attempts to look into the future of the system in terms of anticipated missions and new opportunities. It indicates the client's long-term direction for the system, including both the current system under development and a view of its evolution and eventual replacement.

These three products, the needs, the vision statement, and the goals statement, are "owned" by the client. It is critical that the

client fully subscribe to – and preferably write – the vision and goals statements, and the needs should originate with the client. The formalization of the needs, goals and vision are the primary inputs to the architectural description. The next section describes the elements of that description.

Architectural Elements

> A software architecture is "a set of architectural (or, if you will, *design*) elements that have a particular form."
>
> – Dewayne Perry and Alex Wolf [14]

We wish to distinguish architecture both from requirements and design. Unlike requirements, which exist in the "problem space," architectures reside in the "solution space" of systems to be built. Whereas a requirements specification avoids making decisions as to how a system is to be realized, an architecture does make specific choices pertaining to the form of a solution.

It is useful to view an architecture as a decision making tool or as comprised of decisions. The key to effective decision-making is making the right decision at the right time, no more, no less. Either overspecification or underspecification may lock out future flexibility. Following Burns and Lister [3] (who were writing about design), our method distinguishes:

> a progression of increasingly specific *commitments* ... properties of the system design which [detailed] designers are not at liberty to change. Those aspects of a design to which no commitment is made ... are the subject of *obligations* that lower levels of a design must address. The process of refining obligations into commitments is often subject to *constraints* imposed primarily by the execution environment.

As distinct from a design, however, architectural decisions are not determined solely by design and implementation considerations, but by external ones, too, as described above.

Although it is a consequence of our definition that every system has architecture, that does not imply that all architectures are equal. As systems engineers we are interested in the properties of effective architectures. In our view, an effective architecture shows how to build a system to satisfy clients' needs, in the context of that client's goals and vision. To do this, we need a means of representing architectural elements in useful fashion.

Architectural Representation

As noted above, an architecture is abstract – it may exist only as an idea or conception in someone's head. That is not good enough for engineering purposes. One focus of our work has been to develop techniques for representing architectural information for use by clients, implementors, and other stakeholders. To do this we have adopted the following representation principles:

- An architecture is documented as a model.
- A model is comprised of one or more views.
- Each view is modeled in terms of components, connections and constraints.
- Views are integrated to enable understanding and enforce consistency of the architecture model.

```
view ::=
        view_name,
        vantage_point
        {, component}+,
        {, connection}+,
        {, constraint}+
vantage_point ::= purpose, scope
model_entity ::=
        component | connection | constraint
component ::=
        component_name
        [, component_description]
        [, attribute_part]
connection ::= arrow
constraint ::= arrow
arrow ::=
        label, source, target
        [, description]
        [, attribute_part]
rule ::=
        statement, rationale,
        decision,
        needs_ref,
        exception_part
```

Table 2: Fragment of a Grammar for Architectural Models

To implement these principles, we are developing a formal specification language for architecture representation. Our architecture language has both a formal syntax and graphical counterpart (not shown here). A fragment of the grammar for that language is shown in table 2. The grammar will define the syntax and semantics of models and architectural entities: components, connections and constraints, views, rules and the attributes needed to record an architecture and manage compliance to it.

Views. Building architects do not attempt to capture all salient characteristics of a building in a single model. Instead, they develop a number of renderings or views. Each view shows the whole structure from a particular vantage point. Vantage points are chosen to highlight key concerns. The views may be presented in a variety of media. In the sketching stage, the architect freely draws various views. Later, when blueprints are drawn up as specifications to the builder, there are well-defined relationships between the views.

Adopting the building metaphor, we have found it useful to develop architectural models by separating out distinct *views* of the model. Each view is motivated by a particular concern and with the intent of showing one particular aspect of the system, to answer questions like:

- How should legacy systems be adapted or reused?
- Should data be replicated or centralized?
- How will data storage systems meet security requirements?
- What forms of distribution should new applications support?
- Should a common tool set and methodology be prescribed?

For example, a data-intensive, enterprise-wide information system might benefit from a *data view*. Such a view is concerned with how data is structured, organized and related. It is not a data model – which would be developed during design. Rather, a data view

would address enterprise-wide requirements on data, integration rules for new and legacy data models, distribution and replication policies, etc.

A key principle of views is completeness: for each view, that view is intended to cover the whole system from the selected vantage point. Another principle is that views are not mutually exclusive; rather, a cohesive understanding of the system is gained through integration of interrelated views.

It is tempting to prescribe predefined views: perhaps, functional, data, and physical views – other techniques have taken this approach. However, we do not yet have enough experience to prescribe these, or any, views for all systems. Sometimes, a system's most critical architectural concerns fall outside this familiar set. Instead, we make the selection of views a key step (step 2 of table 1) of the architect's work [10].

Components, Connections and Constraints. Architectural views are described in terms of three primitive constructs: components, connections and constraints [18]. Pursuing the analogy with architectural rendering, these are primitives of expression – not of design.

Components represent the major elements of a view.

Connections represent the major relations between components. They may be behavioral or structural. Connections include "runtime" relationships like control or data flow, but are not limited to these.

Constraints represent laws the system must observe. Constraints apply to components and connections. Constraints can be use to express performance and non-functional characteristics, style and protocol rules, and "laws of nature" which limit the use of resources.

In our representation all three types of primitives are first-class constructs.

Meta Model. We have adopted a uniform representation for views, based on these three primitives. Uniformity is obtained by defining logical relationships between the three primitives. This gives the architect an "open semantics" of systems to work with: the architect may select an existing notation (with its semantics) or devise a new one – as long as the the notation meets the minimum requirements of this "meta model." Our architecture specification language will exploit the meta model to permit users to extend the representation framework for special purposes, via subclassing and related operations. Another advantage of the meta model, which we have not exploited to date, is the prospect for automated support.

View Integration. An important part of architecture development is the establishment of consistency across views. We do this via traceability mechanisms. There are several levels of traceability defined by our method.

Once a view is syntactically checked, using the meta model, it may be integrated with others. View integration is a necessary step in a method like ours because of the nature of views – integration is a meaningful semantic act, not a syntactic projection or transformation which could be automated. The technique we use for view integration is based on Ross' "model tie process" from Structured Analysis (SADT) [17]. Using the tie process, we systematically cross-reference related entities from distinct views, thereby addressing what Shaw calls the "multiple views problem" [19].

Once the views have been integrated, the (resulting) architectural description is reconciled against the original needs analysis (step 5). In this step we ascertain that the architecture covers the stated needs, goal and vision. It is not necessary that each statement of goals, vision, and needs be individually satisfied by the architecture description. However, it is necessary that all questions or issues raised by goals, vision or needs are either identified and resolved by the architecture, or that their resolution is explicitly delegated to the design.

Later in the development, there is a further level of traceability, that of establishing conformance of the design and/or implementation with the architecture (step 7).

Implications for Systems Engineering

The preceding method suggests a very different way to formalize systems engineering. We would argue that any foundation for systems engineering must recognize and incorporate a means for dealing with multiple views. For large systems, it is simply not feasible to construct a single model to represent all systems engineering-relevant information about a system. We adopt the engineering principle of "separation of concerns" to motivate the presentation of an architecture through the construction of multiple views. By developing separate views, each organized around a well-defined vantage point, one can clearly isolate and present major concerns, thereby reducing the perceived complexity of the overall system. The views tend to be independent but interrelated: some details will be found only in one view, others will span two or more views.

Of course, the introduction of multiple views also introduces Shaw's multiple views problem (see above). Our approach deals with this in three ways:

1. Uniform representation (components, connections and constraints) across views
2. "Semantic unification" of the relationships among representation entities via a single meta model
3. Formalized traceability mechanisms for the integration of views and their content

Uniform Representation. In our work, we have concluded that architectural information can be captured by entities of three basic classes, which we can understand formally. This is the basis for our foundational thinking. The foundation afforded by the architecture metaphor is essentially descriptive: the basic laws of architecture are the basic laws of description for the purpose of understanding. Rather than reduce systems engineering concepts to some mathematical construct (whether sets, functions state machines or objects), our architectural approach builds in these elements as first-class constructs, to meet the needs and concerns of the multiple audiences involved in systems architecture. Once these are in place, we can apply other engineering principles to them.

The idea of foundations based on descriptive uniformity is not new with us – although this application is. Similar ideas may be found in the earliest work on Structured Analysis [16, 17].

Unified Meta-model. Consistency is maintained between these multiple views, while retaining their individual semantic integrity, by adopting a uniform meta model of components, connections and constraints. This model can be formalized to the desired degree to support architecture analysis techniques, model verification and validation and other kinds of automated tool support. This is the subject of our current work.

We call this a "meta-model" because (1) it does not prescribe how architectural information is represented, as long as those representations minimally conform to the meta-model; and, (2) a representation's semantics is "open" – determined as much by the

"labels" associated with the entities as the overall graph structure of its components, connections and constraints.

Summary

There is currently significant interest in developing system architecture to control and manage large systems. We have been developing a practical architecture method to manage the architectural process, based on a traditional building architecture metaphor. The basis of this metaphor is the definition of the term "architecture" and the role of architecture in system construction. We have furthered the metaphor by articulating the role and activities of an architect in system development.

In this paper, we have argued that the architectural metaphor has wider applicability to system engineering by providing a non-reductionist, description-based framework for understanding system engineering activities and products giving equal weight to the multiple concerns of diverse participants.

Acknowledgments. The work described here was performed by the MITRE Software Center (Bedford). Initial applications of the practical architecture method were sponsored by the U.S. Army Information Systems Engineering Command (ISEC). Contributors to the definition and application of the method include: R. Baldwin, C. Byrnes, D. Emery, R. Harris, K. Heideman, J. Hustad, M. Kinnebrew, T. Nixon, J. Moore, S. Schwarm, and D. Waxman.

References

[1] R. Samuel Alessi et al. The foundation of systems engineering. In *Proceedings of the Fifth Annual Symposium of the International Council on Systems Engineering*, 1995. St. Louis, Missouri, July 22-26, 1995.

[2] Barry W. Boehm. Software process architectures. In *Proceedings of the First International Workshop on Architectures for Software Systems*, Seattle, WA, 1995.

[3] A. Burns and M. Lister. A framework for building dependable systems. *The Computer Journal*, 34(2), 1991.

[4] Defense Information Systems Agency. *Technical Architecture Framework for Information Management (TAFIM)*, version 2.0 edition, 1995.

[5] L. Druffel, N. E. Loy, R. A. Rosenberg, R. J. Sylvester, and R. A. Volz. Information architectures that enchance operational capability in peacetime and wartime. Technical report, US Air Force Science Advisory Board, February 1994.

[6] David E. Emery and Richard F. Hilliard. "Architecture," methods and open issues. In D. Garlan, editor, *Proceedings of the First International Workshop on Architectures for Software Systems*, Seattle, WA, 1995. Published as CMU–CS–TR–95–151.

[7] David E. Emery, Richard F. Hilliard, and Timothy B. Rice. Experiences applying a practical architectural method. In Alfred Strohmeier, editor, *Reliable Software Technologies – Ada-Europe '96*, number 1088 in Lecture Notes in Computer Science. Springer, 1996.

[8] Cristina Gacek, Ahmed Abd-Allah, Bradford Clark, and Barry W. Boehm. On the definition of software system architecture. In *Proceedings of the First International Workshop on Architectures for Software Systems*, Seattle, WA, 1995.

[9] Richard F. Hilliard. Comments on kogut and clements. email.

[10] Richard F. Hilliard. The notion of 'architecture' in model-based software engineering. In *Proceedings of the Workshop on Domain-Specific Architectures*, Hidden Valley, PA, July 1990.

[11] Barry M. Horowitz. The importance of architecture in DOD software. Technical Report M 91-35, The MITRE Corporation, July 1991.

[12] Ralph E. Johnson and Vincent F. Russo. Reusing object-oriented designs. Technical Report UIUCDCS 91–1696, University of Illinois, 1991.

[13] A. H. Levis and D. M. Perdu. A systems engineering approach to information architecture design. In *Proceedings 1994 IFAC Symposium on Integrated Systems Engineering*, 1994.

[14] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Sofware Engineering Notes*, 17(4), October 1992.

[15] Charles Rattray. The shape of complex systems. In *Computer Aided Systems Theory – EUROCAST '93: a selection of papers from the Third International Workshop on Computer Aided Systems Theory*, 1993. Las Palmas, Spain.

[16] Douglas T. Ross. Structured Analysis (SA): a language for communicating ideas. *IEEE Transactions on Software Engineering*, SE-3(1), January 1977.

[17] Douglas T. Ross. Removing the limitations of natural language (with the principles behind the RSA language). In Herbert Freeman and Philip M. Lewis II, editors, *Software engineering: proceedings of the Software Engineering Workshop held in Albany, Troy, and Schenectady, New York, from May 30–June 1, 1979*. Academic Press, New York, 1980.

[18] Thomas F. Saunders, Barry Horowitz, and Matt L. Mleziva. New process for acquiring software architecture. Technical Report M 92B0000126, The MITRE Corporation, 1992.

[19] Mary Shaw. Comparing architectural design styles. *IEEE Software*, 28(11):27–14, November 1995.

[20] Mary Shaw and David Garlan. An introduction to software architecture. In V. Ambriola an G. Tortora, editor, *Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing Company, 1993.

[21] Bernhard Thomé. *Systems engineering: principles and practice of computer-based systems engineering*. Wiley, 1993.

Biography

Richard F. Hilliard II is a Lead Engineer at The MITRE Corporation in the Open Systems Engineering Group. Since 1982 he has worked on software technology and command and control applications at MITRE. From 1991–1994 he was a member of the Ada 9X Mapping/Revision Team, which defined the ISO standard for Ada 95. Rich is an officer of the League for Programming Freedom.

Timothy B. Rice is a Lead Information Systems Engineer and supervises the Open Systems Engineering Group in the MITRE Bedford Software Center. He is experienced in Open Systems and Client-Server architectures for the Command and Control, Information Systems and Medical Instrumentation domains. Tim's interests are in software architecture and frameworks, software development,

object-oriented design, software design evaluation, open system definition and application of open systems environments, software tools and programming environments, and software system cost estimation.

Stephen C. Schwarm is a Principal Engineer at The MITRE Corporation in the Division for Integrated Air Operations C^3. He currently supports the Theater Battle Management Core Systems project as the Chief Engineer. Steve is also Chair of the Language Bindings Working Group of the POSIX Standards Committee. He is a member of the Association of Computing Machinery and an IEEE/Computer Society affiliate.